EECS 280 – Lecture 6

Strings and I/O

https://eecs280staff.github.io/notes/05_Strings_Streams_IO.html

Announcements

- P2 is out, due Friday 2/4 (~1 week)
  - Can work through Matrix component now
  - Should have everything else after today's material
- Remember to fill stuff out!
  - CARES survey (0.5% of your total grade) **TONIGHT!**
  - Exam accommodations form
  - Exam conflict forms 1/28
- Lab 2 this week
  - Due Sun 8 pm

Agenda

- **Finish up Compound Objects**
- Strings
  - C-Style strings
  - C++ strings
- Command Line Arguments
  - argv and argc
- Streams

Review

- Structs contain multiple elements, which we access using "." operator

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person p1 = { 17, "Kim", true };
  Person p2 = { 17, "Ron", true };

  p1.isNinja = false;
}
```

Review

- Structs contain multiple elements, which we access using "." operator

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  Person p1 = { 17, "Kim", true };
  Person p2 = { 17, "Ron", true };

  p1.isNinja = false;
}
```

Review

- If we pass a pointer to a struct, must first "*" and then use "."
  - Or use shortcut "->" (two characters "-" and ">")

need to dereference at first

```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  (*p).age += 1;
  if ((*p).age > 18) {
    (*p).isNinja = true;
  }
}
```

Because the compiler don't know how to take the subelement of an address

is equivalent to…

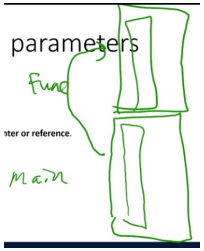
```
// REQUIRES: p points to a Person
void Person_birthday(Person *p) {
  p->age += 1;
  if (p->age > 18) {
    p->isNinja = true;
  }
}
```

Passing `structs` as parameters

- You **usually don't** want to pass by value.
  - Might be very large!

```
void func(Person p);
```

spending memory and time to copy the data.



- If you intend to **modify** the outside object, pass by **pointer or reference**.

```
void func(Person *p);
void func(Person &p);
```

- Otherwise, pass by **pointer-to-const** or **reference-to-const**. (Safer and more efficient than by value.)

```
void func(Person const *p);
void func(Person const &p);
```

Exercise

```
struct Person {
  int age;
  string name;
  bool isNinja;
};

int main() {
  int x;
  Person alex;
  alex.age = 20;
  Person jon;

  Person *people[] = {
    &alex, &jon
  };

  // print Alex's age
  cout << _____________ ;
}
```

read from right to left

pointer

arry

(*people)->age ✓

Poll:

Which line(s) of code can go in the blank to print Alex's age?

first element of the array

A) (*people).age

B) *(people->age)?

C) people[0].age

D) people[0]->age

Agenda

- Finish up Compound Objects
- **Strings**
  - C-Style strings
  - C++ strings
- Command Line Arguments
  - argv and argc
- Streams

Strings

- Technically, we don’t need anything new to represent text
  - Just use an array of chars and go from there
- But… text processing is so common, C and C++ both added data types to make more streamlined
  - C-style strings: arrays with special rules
  - C++ strings: custom data types with special functions

C-Style Strings

- C string is a special type of array
- Special how?
  1. The array elements are of type ‘char’
  2. The last element of the array is always ‘\0’ (null character)
  3. Special syntax for initializing
  4. Special rules for printing

C-Style Strings

"\0" tells compiler ; this is the end. don't pass this.

So in C++. we need use parameter → the length of the array.

but C doesn't need.

- The array elements are of type 'char'

```
char str1[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

- C-strings have special initialization rules

```
char str2[6] = "hello";
```

Compiler automatically puts '\0' at the end of string literals.

- The last element of the array is always null character
  - `'\0'` in code
  - ASCII value `0`
  - Acts as a **sentinel** to say "Whoa, the string stops here!"
- Of course, character arrays turn into pointers as well.

```
char *strPtr = str1;
```

C-Style Strings as `char` Arrays

- char values are just numbers underneath

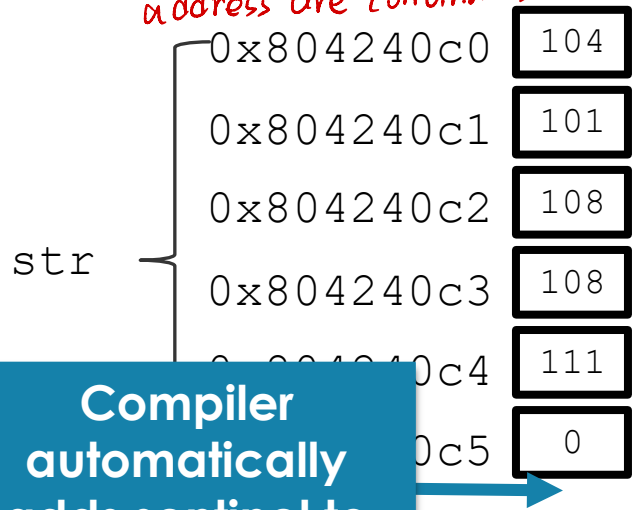
ASCII Codes

Symbol	Number
'\0'	0
...	
'e'	101
'f'	102
'g'	103
'h'	104
...	

Null character is the sentinel. It has value 0.

```
char str[6] = "hello"
```

address are continuous



Compiler automatically adds sentinel to string literals.

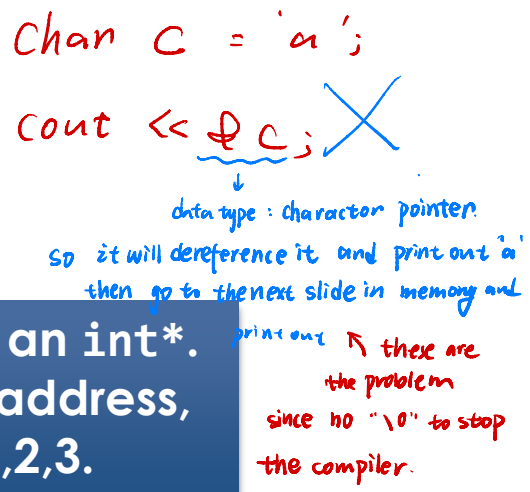
Sentinels

Poll:

Why didn't we use sentinels in other arrays?

a) Purely a convention
b) Performance would be worse
c) Can’t guarantee it will work correctly
d) I have no idea!

C-Style Strings and cout



- We saw earlier you can't print out arrays.

```
int array[3] = { 1, 2, 3 };
cout << array << endl;
```

int* → address

Turns into an int*. Prints an address, not 1,2,3.

- But you can print out C-style strings.

```
char str[6] = "hello";
cout << str << endl;
```

special char* will dereference it and add the pointer to go to the next address until meet "\0"

Turns into a char*. Prints out "hello".

- cout treats ALL char* as C-style strings
  - Starts printing characters until it finds a null character.
  - Don't try to print a char* not pointing into a C-style string!

C-Style Strings and cout

Poll:

const char *str = “hello”;

Which of the following prints a single character (select all that apply)?

a) cout << str;

b) cout << str[1];

c) cout << &str[1];

d) cout << &str[4];

str[i] → *(str+i)

data type: pointer to a character chain*

hello

datatype: single character chain.

e

Str is an array of characters

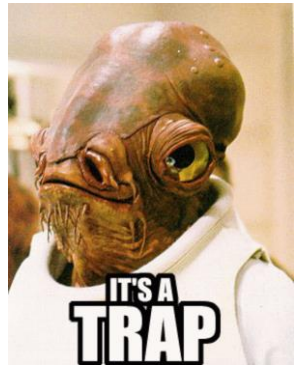
ello

pointer to a character chain*

o

Be Careful with C-Style Strings

- This code doesn't do what it first appears to. Remember, they turn into pointers.



Actually tests if at the same address.

Doesn't compile. Type mismatch.

Makes ptr point to different string.

```
char str1[6] = "hello";
char str2[6] = "hello";
char str3[6] = "apple";
char *ptr = str1;

// Test for equality?
str1 == str2;
// Copy strings?
str1 = str3;
// Copy through pointer?
ptr = str3;
```

Working with C-Strings

- If we can't use simply operators (=, ==) to work with c-strings, we'll need to rely on loops
- C standard library provides many functions to do this for you
  - strlen() - returns length of c-string (not including \0)
  - strcpy() - copies c-string from one location to another
  - strcmp() - compare two strings
- But you should know how these functions work!

Traversing a C-Style String

Lobster:
L05.1_strlen

```
char str[6] = "hello";
cout << strlen(str) << endl; // Prints 5
```

- Just keep going until we find the **sentinel**.
  - When the current element has value `'\0'`

```
int strlen(const char *str) {
  const char *ptr = str;
  while (*ptr != '\0') {
    ++ptr;
  }
  return ptr - str;
}
```

Pointer starts at beginning of the string.

Continue until sentinel value is found.

Increment pointer.

Take difference to see how many steps we took. (Does not count '\0'.)

Extra Exercise: `strcpy`

NOTE: if you set a pointer to src, it needs to be delared as "const char *"

- Find the file "`L05.2_strcpy`" on Lobster.

lobster.eecs.umich.edu

```
char word1[5] = "frog";
char word2[7] = "lizard";
strcpy(word2, word1);
cout << word2; // should print "frog"
```

- Write the function `strcpy`.
- `main` already contains a few tests.
- Use traversal by pointer.
  - This is customary for working with C-style strings.

Solution: strcpy

```
char word1[5] = "frog";
char word2[7] = "lizard";
strcpy(word2, word1);
cout << word2; // should print "frog"
```

```
void strcpy(char *dst, const char *src) {
  while (*src != '\0') {
    *dst = *src;
    ++src;
    ++dst;
  }
  *dst = *src;
}
```

Assign character value from *src to *dst.

We'll be using src and dst to march through the arrays.

Increment pointers.

Finally, copy null character.

Comparing C-strings

- Don’t use built-in operators.
  These will just compare addresses.
- Instead, use the `strcmp` function.
- `strcmp(A,B)` returns:

  `negative` if A less than B

  `0` if A equal to B

  `positive` if A greater than B

What about C++ strings?

	C-Style Strings	C++ Strings
Library Header	`<cstring>`	`<string>`
Declaration	`char cstr[];` `const char *cstr;`	`string str;`
Length	`strlen(cstr)`	`str.length()`
Copy value	`strcpy(cstr1, cstr2)`	`str1 = str2`
Indexing	`cstr[i]`	`str[i]`
Concatenate	`strcat(cstr1, cstr2)`	`str1 += str2`
Compare	`strcmp(cstr1, cstr2)`	`str1 == str2`

string to C-style string: const char *cstr = str.c_str();

C-style string to string: string str = string(cstr);

Agenda

- Finish up Compound Objects
- **Command Line Arguments**
  - **argv and argc**
- Streams
- Strings
  - C-Style strings
  - C++ strings

Agenda

- Finish up Compound Objects
- Strings
  - C-Style strings
  - C++ strings
- **Command Line Arguments**
  - argv and argc
- Streams

Command Line Arguments

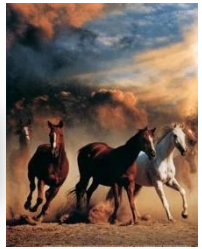
- Some programs, like stats_tests.exe in P1, do the same thing every time you run it

```
$ ./stats_tests.exe
All tests pass!
```

- If you want the program to do something different, you need to change the source code and recompile

Command Line Arguments

- But other programs behave differently depending on how you evoke the program
  - E.g. project two, we might run it to resize a picture of horses



```
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250
```

or a picture of a face

```
$ ./resize.exe Beaumont.ppm oh_god.ppm 400 250
```

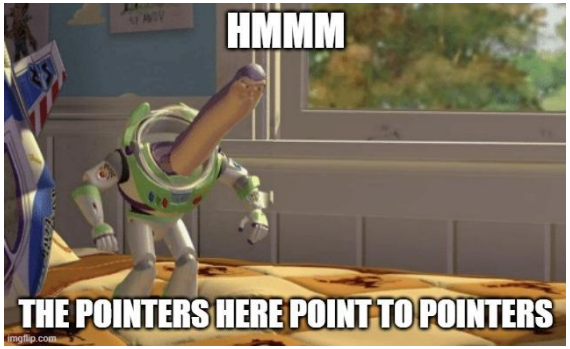
Command Line Arguments

```
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250
```

- `resize.exe` is the name of the program to run.
- The other "words" are **arguments** to the program.
  - The **shell** (a.k.a. terminal, console, etc.) starts the program and passes arguments.
  - The program gets the arguments. In C++, they are passed as parameters to `main`.
- **How can we write programs to take arguments like this?**

argv and argc

- Two parameters to main:
  - argc – the number of arguments
  - argv – an array of the arguments
- argv is an **array of C-style strings**.



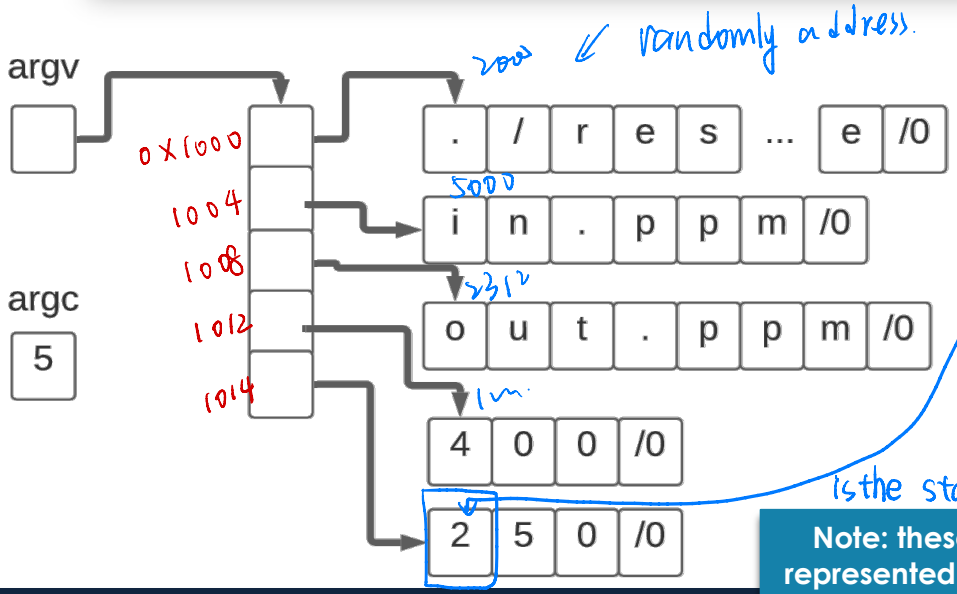

```
int main(int argc, char *argv[]) {

}
```

array of character pointer

Compiler turns this into char **argv.

argv and argc

```
$ ./resize.exe in.ppm out.ppm 400 250
```



Poll

- What is the data type of argv[4]? character pointer
- What is the value?
- What does `cout << argv[4]` print? 250

Note: these are all chars represented by ASCII values

atoi

- Arguments are always passed in as c-strings
  - What if we want to treat it like an integer?
    - Like width of PPM file in P2?
  - Doing arithmetic directly on argv[] results in pointer arithmetic… not what we want

```
int main(int argc, char *argv[]) {
  cout << argv[1] << " + 1 = “
       << argv[1] + 1;
}
```

argv[1] is char*, argv[1] + 1 is calculating a new address

```
$ ./program 53
53 + 1 = 0x1001
```

atoi

- Arguments are always passed in as c-strings
  - If we want to treat them like integers, we need to use "atoi"

```
#include <cstdlib>
// EFFECTS: parses s as a number and
//          returns its int value
int atoi(const char *s);
```

```
int main(int argc, char *argv[]) {
  cout << argv[1] << " + 1 = “
       << atoi(argv[1]) + 1;
}
```

```
$ ./program 53
53 + 1 = 54
```

Agenda

- Finish up Compound Objects
- Strings
  - C-Style strings
  - C++ strings
- Command Line Arguments
  - argv and argc
- **Streams**

Streams

- An abstraction that allows you to read/write data from input/output

Reading and Writing Images in PPM Format

The `Image` module also provides functions to read and write `Image`s from/to the PPM image format. Here's an example of an `Image` and its representation in PPM.

Image	Image Representation in PPM
	P3 5 5 255 0 0 0 0 0 0 255 255 250 0 0 0 0 0 0 255 255 250 126 66 0 126 66 0 126 66 0 255 255 250 126 66 0 0 0 0 255 219 183 0 0 0 126 66 0 255 219 183 255 219 183 0 0 0 255 219 183 255 219 183 255 219 183 0 0 0 134 0 0 0 0 0 255 219 183

Streams

- Example input streams:
  - Standard input
    - a.k.a “cin”
    - reads from terminal
  - ifstream
    - input file stream
    - read from a file
- Read using “extraction operator” >>

- Example output streams:
  - Standard output
    - “cout”
    - writes to terminal
  - ofstream
    - output file stream
    - write to a file
- Write using “insertion operator” <<

cin Example

- We're already familiar with reading input from standard input (`cin`).

words.cpp

```
string word;
while (cin >> word) {
  cout << "word = '" << word << "'" << endl;
}
```

Will stop when an "end of file" character is read. To type this at the console, use ctrl+d.

```
$ g++ words.cpp -o words

$ ./words
hello world!
word = 'hello'
word = 'world!'
goodbye
word = 'goodbye'
```

File I/O with Streams

- In C++, we can read and write files directly with `ifstream` and `ofstream` objects

```
#include <fstream>
```

- `ifstream` and `ofstream` allow you to…
  - …read a file just like reading from `cin`
  - …write to a file just like printing to `cout`

File Input: `ifstream`

```
int main() {
  string filename = "hello.txt";
  ifstream fin;
  fin.open(filename);
  if (!fin.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  string word;
  while (fin >> word) {
    cout << "word = '" << word << "'" << endl;
  }
  fin.close();
}
```

Open a file using `fin` variable

Check for success opening file.

Read one word at a time and check that the read was successful.

File Output: ofstream

```
int main() {
  const int SIZE = 4;
  int data[SIZE] = { 1, 2, 3, 4 };
  string filename = "output.txt";
  ofstream fout;
  fout.open(filename);
  if (!fout.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  for (int i = 0; i < 4; ++i) {
    fout << "data[" << i << "] = " << data[i] << endl;
  }
  fout.close();
}
```

output.txt

```
data[0] = 1
data[1] = 2
data[2] = 3
data[3] = 4
```

Using istream/ostream generically

- ofstream instances and cout are both instances of the more generic "ostream"
  - And ifstream instances and cin are both instances of "istream"
  - We can write generic functions that work for either (used in P2!)

```
void print(ostream &os) {
  os << "hi" << endl;
}
int main() {
  ofstream fout;
  fout.open("output.txt");
  print(fout);  // prints to output.txt
  print(cout);  // prints to terminal
}
```

Next Time

- Abstract Data Types
  - Why do we design the functions the way we do in P2?

- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

Extra Slides

- You don't need to know these for class

Aside: Prefix vs. Postfix Increment

- Statements in a programming language can have **evaluations** and **side-effects**
  - The evaluation is what you see if you printed the statement
  - The side effect is whatever variables get updated

```
int x = 1;
cout << x + 1;      // evaluation of x+1 is 2
                    // has no side effect
cout << x = x + 1; // evaluation of x=x+1 is also 2
                    // has side effect of incrementing x
```

Prefix vs. Postfix Increment

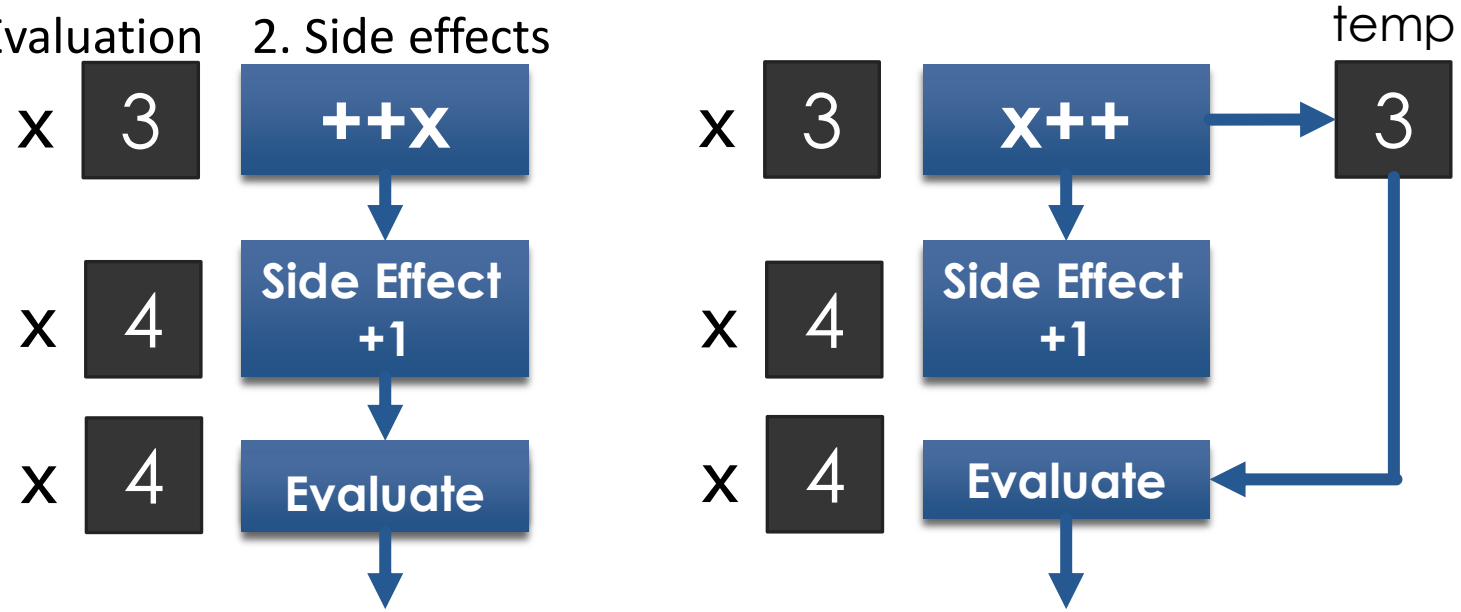
- Prefix and postfix have the same **side effects** (incrementing the variable by 1), but different evaluations
  - **x++** evaluates to the **old** value (needs to keep track of it via a temporary variable
  - **++x** evaluates to the **new** value (does not need a temp)

```
int x = 1;
cout << x++; // prints 1
cout << x;   // prints 2
```

```
int x = 1;
cout << ++x; // prints 2
cout << x;   // prints 2
```

Prefix vs. Postfix Increment

- Parts of an expression



For fun: strcpy (Cute Version)

```
char str1[6] = "hello";
char str2[6] = "apple";
strcpy(str1, str2); // str1 array now holds "apple"
```

```
void strcpy(char *dst, const char *src) {
  while (*dst++ = *src++);
}
```

Condition for loop depends on value that was assigned to `*dst`. `'\0'` turns into false.

Assignment evaluates to value that was assigned.

Dereference is applied to old addresses, and character is copied.

Postfix increment moves both pointers, but evaluates to old values (addresses).